15-750 Algorithms (Spring 2002) **Prof. Manuel Blum**

Homework #2 **Solutions**

Problem 1: Minimum spanning tree without union-find (30 points)

Let $G = (V, E)$, $|V| \geq 2$, be a connected (weighted) graph with *distinct* positive integer weights on the edges.

(a) Is the following assertion true? If so, prove it. If not, give a counterexample.

> **Assertion:** For every vertex $v \in V$, the (unique) edge e of minimum weight that is incident to v is necessarily in any minimum spanning tree of G.

The assertion is true.

Proof. Assume that a minimum spanning tree T of G does not contain the minimum weight edge e incident to v, for some $v \in V$, say $e = (v, u)$. Since T is a spanning tree, it contains a path P connecting u and v, which contains some edge $e' \neq e$ incident to v, say $e' = (v, w)$. By assumption, the weight of e' is larger than the weight of e. It is easy to see, that $T' = (T \cup \{e'\}) \setminus \{e\})$ is also a spanning tree of G. Moreover, the weight of T' is smaller than the weight T, which contradicts our assumption, that T is a minimum spanning tree. □

(b) In the case under consideration in which no two edges of G have the same weight, can G have more than one minimum spanning tree? (Naturally, you should prove your answer.)

No, such a graph G cannot have more than one minimum spanning tree.

Proof. Assume that G has two different minimum spanning trees T_1 and T_2. Consider the edges in both trees sorted by their weight, and take the first (the lightest) edge $e = (u, v)$ contained in one of the trees and not contained in the other. Wlog. assume $e \in T_1$. Inserting e to T_2 would produce a cycle, which contains an edge $e' = (u', v')$ not present in T_1 (otherwise T_1 wouldn't be a tree). Since e is the lightest edge in which both trees differ, e' has larger weight than e. So $T' = (T_2 \cup \{e\}) \setminus \{e'\}$ is a spanning tree of G with weight smaller than T_2, which contradicts our assumption, that T_2 is a minimum spanning tree. □

(c) Describe how to use heaps to implement a minimum spanning tree algorithm in $\mathcal{O}(m + n \log n)$, where $n = |V|$ and $m = |E|$.

Use *ordinary binary heaps* if possible; if not (because efficiency is compromised), then use *binomial heaps* — in this case explain why ordinary binary heaps won't do.

Try to make it with binomial heaps, but if you won't succeed then use *Fibonacci heaps* — in this case explain why binomial heaps won't do.

(d) Analyze the running time of your algorithm. Your algorithm should run in time $\mathcal{O}(m + n \log n)$.

We use Prim's algorithm (cf. Cormen *et al.* "Introduction to Algorithms", Section 24.2), which incrementally "grows" a tree T by starting from any node and by adding to T the minimum weight edge connecting T to some vertex not yet in T.

The key to implementing this algorithm efficiently is to find a way to select the new edges. For this we use a priority queue containing all vertices not yet in T, where the priority of a vertex v is determined by the minimum weight of any edge connecting v to a vertex in T. The queue can be implemented as a (min) heap. However, to achieve the required bound we need a very efficient (amortized $\mathcal{O}(1)$ cost) `decrement` operation, which is used to update the priorities in the queue after deleting the current minimum. Unfortunately, both ordinary and binomial heaps give only $\mathcal{O}(\log n)$ bound on `decrement`, so we have to use Fibonacci heaps to accomplish the task. (see the textbook for details)

Problem 2: Yet another amortized analysis (20 points)

(this problem is due to Danny Sleator)
Start with an undirected graph that consists three vertices in the form of a triangle. A sequence of two types of operations are performed on the graph.

link(i, j)**:** Edge (i, j) is added to the graph. The cost of this operation is one.

chop(i)**:** Vertex i (of degree d) is deleted from the graph, and replaced by a collection of vertices $i_1, i_2, \ldots, i_d$. These vertices are connected in a cycle, and also each of the former neighbors of i are attached to exactly one of $i_1, \ldots, i_d$. After this step, each i_j is of degree 3. The cost of this operation is d, the original degree of i.

Find constants c_l and c_c (the smallest you can) such that the total cost of a sequence of n links and m chops is at most $c_l n + c_c m$.

(Solution due to Jernej Barbic, Vincent Conitzer, Vahe Poladian)
Say $c_l = 3$, $c_c = 3$. Upon a link, we store one credit with each of the vertices it is linking. Then it is easy to see that a node of degree d always has at least $d - 3$ credits (since a newly generated node has degree 3, and the only way to increase the degree of a vertex is by linking something to it). Hence, these credits plus the 3 we get with any chop can pay for the chop at that node.

To show that each of c_l, c_c needs to be at least 3 (independent of what the other is!), consider first a sequence of only chops, the first one of degree 2, and each one after that on one of degree 3. Thus if the sequence has length m it costs $3m - 1$. $3m - 1$ divided by m goes to 3 as m goes to infinity, so we need at least 3 credits per chop.

For the links, consider the sequence of n operations link(a, b), followed by chop(a) and chop(b). This costs $n + 2(n + 2) = 3n + 2$. The two chops can only pay for $2c_c$, so the links need to pay $3n + 2 - 2c_c$, i.e. $3n + 2 - 2c_c$ divided by n per link. Whatever (constant) c_c is, this goes to 3 as N goes to infinity, so we need at least 3 credits per link.